Parallel Deep Neural Network Training for Big Data on Blue Gene/Q

I-Hsin Chung Tara N. Sainath Bhuvana Ramabhadran Michael Picheny
John Gunnels Vernon Austel Upendra Chauhari Brian Kingsbury

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{ihchung,tsainath,bhuvana,picheny,gunnels,austel,uvc,bedk}@us.ibm.com

***Abstract*—Deep Neural Networks (DNNs) have recently been shown to significantly outperform existing machine learning techniques in several pattern recognition tasks. DNNs are the state-of-the-art models used in image recognition, object detection, classification and tracking, and speech and language processing applications. The biggest drawback to DNNs has been the enormous cost in computation and time taken to train the parameters of the networks - often a tenfold increase relative to conventional technologies. Such training time costs can be mitigated by the application of parallel computing algorithms and architectures. However, these algorithms often run into difficulties because of the cost of inter-processor communication bottlenecks. In this paper, we describe how to enable Parallel Deep Neural Network Training on the IBM Blue Gene/Q (BG/Q) computer system. Specifically, we explore DNN training using the data-parallel Hessian-free 2nd order optimization algorithm. Such an algorithm is particularly well-suited to parallelization across a large set of loosely coupled processors. BG/Q, with its excellent inter-processor communication characteristics, is an ideal match for this type of algorithm. The paper discusses how issues regarding programming model and data-dependent imbalances are addressed. Results on large-scale speech tasks show that the performance on BG/Q scales linearly up to 4096 processes with no loss in accuracy. This allows us to train neural networks using billions of training examples in a few hours.**

Keywords*—Big Data, Speech Recognition, High Performance Computing

I. Introduction

Deep Neural Networks (DNNs) have become a popular in the machine learning community over the last few years [1], showing significant gains over state-of-the-art machine learning methods used in both speech and image processing. The development of pre-training algorithms [2] and better forms of random initialization [3], as well as the availability of improved hardware, has made it possible to train deeper networks than before, and in practice these deep networks have achieved excellent performance in speech recognition applications [4], [5], [6].

However, one drawback of DNNs is that training remains very slow, particularly in applications involving a lot of data. Given the increased amount of data currently available, particularly from multimedia sources, the need for algorithms to run on large-scale tasks is critical. Slow DNN training can be attributed to a variety of causes. First, models for big-data tasks are often trained with millions to billions of training examples, as using increased amounts of training data often improves DNN performance [7]. Second, large data sets often use DNNs with a large number of parameters (i.e., can be 10-50 million DNN parameters in many speech tasks) [8], [6]. Third, to date the most popular methodology to train DNNs is the first-order stochastic gradient descent (SGD) optimization technique, which is a serial algorithm executed on a multi-core CPU. While second order optimization techniques, which are much easier to parallelize across machines, have been explored for DNN training, these methods are not always faster than training DNNs via SGD. However, these methods yield one of the best performing models for several training criteria. For example, as shown in [9], parallelization of dense networks can actually be slower than serial SGD training, typically because of communication costs in passing models and gradients as well as the need to run more training iterations compared to serial SGD. Therefore, parallelization methods for DNNs have not enjoyed much success, and training DNNs via SGD is still the most popular technique.

IBM Blue Gene/Q is designed to deliver ultra-scale performance with a standard programming environment. It is currently one of most efficiently scalable systems for computationally intensive science applications. It has enabled science to address a wide range of complex problems. With its good performance in analytics applications, Blue Gene/Q is a good platform for big data applications.

The objective of this paper is to explore speeding up second order optimization of DNN training using a Blue Gene/Q computer system. With this Parallel DNN training application enablement and performance analysis, it also provides input for future generation system/architecture design. Speech recognition is one important big-data application, which we will use as an example in this paper, in presenting our algorithm to speed up DNN training. Speech recognition is an important, real-world big-data application, as models are large (roughly 10-50 million parameters) and models are typically trained on millions to billions of training examples [6]. Furthermore, the second order Hessian Free training algorithm [10] has been shown to produce state-of-the art performance (measured by word-error-rate) on vision and speech tasks [5], making it an ideal algorithm to explore on a Blue Gene/Q architecture.

The rest of this paper is organized as follows. Section II shows the related work. Section III gives an overview about the Blue Gene/Q system. Section IV introduces the Parallel Deep Neural Network Training. Section V describes the challenges to enable and tune the application on the Blue Gene/Q system. Section VI presents the performance results. The discussion and conclusion are presented in Section VII and VIII.

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.66

II. Related Work

A. DNN Training Methods

Stochastic Gradient Descent (SGD) remains one of the most popular approaches for training DNNs. SGD methods are simple to implement and are generally faster for large data sets when compared to second order methods [9].

While parallel SGD methods have been successfully explored for convex problems [11], for non-convex problems such as DNNs it is very difficult to parallelize SGD across machines. With SGD the gradient is computed over a small collection of frames (known as a mini-batch), which is typically on the order of 100-1,000 for speech tasks [12]. Splitting this gradient computation onto a few parallel machines, coupled with the large number of network parameters used in speech tasks, results in large communications costs in passing the gradient vectors from worker machines back to the master. Thus, it is generally cheaper to compute the gradient serially on one machine using the standard SGD technique [13]. It is important to note that recently [14] explored a distributed asynchronous SGD method to improve DNN training speed.

Second-order batch methods, including conjugate gradient (CG) or limited-memory BFGS (L-BFGS), generally compute the gradient over all of the data rather than a mini-batch, and therefore are much easier to parallelize [15]. However, as shown in [9], parallelization of dense networks can actually be slower than serial SGD training, again because of communication costs in passing models and gradients as well as the need to run more training iterations compared to serial SGD.

B. Improved Matrix Libraries

Math library involving matrix calculation plays an important role in the high performance computing. Prior work such as [16] involves DGEMM on Blue Gene/Q. General optimization of mathematical routines are needed in applications such as [17], [18]. The matrix calculation tuning for Parallel Deep Neural Network Training is focused on SGEMM which is not as heavily utilized in conventional high performance computing applications.

III. Architecture

The IBM Blue Gene/Q system [19] is the third generation of high-performance computer in the Blue Gene series. Blue Gene/Q's processor [20] is a system-on-chip multi-core processor with 16 A2 PowerPC 64-bit cores. Each A2 core runs at 1.6 GHz, is 4-way multi-threaded, and is capable of executing instructions in-order in two pipelines. One pipeline executes all integer control and memory access instructions while the other pipeline executes all floating point arithmetic instructions. The floating point unit is a 4-wide SIMD double precision unit known as the QPX. The QPX unit is capable of delivering up to four fused multiply-add results per processor clock.

Each core has 16K-byte private level 1 (L1) cache and 2K-byte prefetching buffer (L1P). All the cores on the same processor share a 32M-byte level 2 (L2) cache. The L2 cache provides versioning support which is used to implement both hardware transactional memory and thread level speculation. The compute nodes are connected in a 5-D torus network with a total network bandwidth of 44 GB/s per node.

Built on the Blue Gene hardware design is an efficient software stack and operating system. The lightweight compute node kernel (CNK) is designed to be efficient and therefore neither virtual memory nor multi-tasking is supported. The file I/O is offloaded to the dedicated I/O nodes via network communication

IV. Parallel Deep Neural Network Training

The challenge in performing distributed optimization is to find an algorithm that uses large data batches that can be split across compute nodes without incurring excessive overhead while simultaneously achieving performance competitive with stochastic gradient descent. One class of algorithms for this problem uses second-order optimization, with large batches for the gradient and much smaller batches for stochastic estimation of the curvature [10], [21], [22]. A distributed implementation of one such algorithm has already been applied to learning an exponential model with a convex objective function for a speech recognition task [21].

The current study uses Hessian-free optimization [10] because it is specifically designed for the training of deep neural networks, which is a non-convex problem. Let $\boldsymbol{\theta}$ denote the network parameters, $\mathcal{L}(\boldsymbol{\theta})$ denote a loss function, $\nabla\mathcal{L}(\boldsymbol{\theta})$ denote the gradient of the loss with respect to the parameters, $\mathbf{d}$ denote a search direction, and $\mathbf{B}(\boldsymbol{\theta})$ denote a matrix characterizing the curvature of the loss around $\boldsymbol{\theta}$. The central idea in Hessian-free optimization is to iteratively form a quadratic approximation to the loss,

$$\mathcal{L}(\boldsymbol{\theta}+\mathbf{d}) \approx \mathcal{L}(\boldsymbol{\theta}) + \nabla\mathcal{L}(\boldsymbol{\theta})^T\mathbf{d} + \frac{1}{2}\mathbf{d}^T\mathbf{B}(\boldsymbol{\theta})\mathbf{d} \qquad (1)$$

and to minimize this approximation using conjugate gradient (CG), which accesses the curvature matrix only through matrix-vector products $\mathbf{B}(\boldsymbol{\theta})\mathbf{d}$ that can be computed efficiently for neural networks [23]. If $\mathbf{B}(\boldsymbol{\theta})$ were the Hessian and conjugate gradient were run to convergence, this would be a matrix-free Newton algorithm. In the Hessian-free algorithm, the conjugate gradient search is truncated, based on the relative improvement in approximate loss, and the curvature matrix is the Gauss-Newton matrix [24], which unlike the Hessian is guaranteed positive semidefinite, with additional damping: $\mathbf{G}(\boldsymbol{\theta}) + \lambda\mathbf{I}$.

Our implementation of Hessian-free optimization, which is illustrated as pseudo code in Algorithm 1, closely follows that of [10], except that it currently does not use a preconditioner [25]. Gradients are computed over all the training data. Gauss-Newton matrix-vector products are computed over a sample (about 1% to 3% of the training data) that is taken each time `CG-Minimize` is called. The loss, $\mathcal{L}(\boldsymbol{\theta})$, is computed over a held-out set. `CG-Minimize`$(q_{\boldsymbol{\theta}}(\mathbf{d}), \mathbf{d}_0)$ uses conjugate gradient to minimize $q_{\boldsymbol{\theta}}(\mathbf{d})$, starting with search direction $\mathbf{d}_0$. Similar to [10], the number of CG iterations is stopped once the relative per-iteration progress made in minimizing the CG objective function falls below a certain tolerance. The `CG-Minimize` function returns a series of steps $\{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_N\}$ that are then used in a backtracking procedure. The parameter update, $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{d}_i$, is based on an

Armijo rule backtracking line search. $\beta < 1.0$ is a momentum term.

Algorithm 1 Hessian-free optimization (after [10]).

initialize $\boldsymbol{\theta}$; $\mathbf{d}_0 \leftarrow \mathbf{0}$; $\lambda \leftarrow \lambda_0$; $\mathcal{L}_{\text{prev}} \leftarrow \mathcal{L}(\boldsymbol{\theta})$
while not converged **do**
  $\mathbf{g} \leftarrow \nabla\mathcal{L}(\boldsymbol{\theta})$
  Let $q_{\boldsymbol{\theta}}(\mathbf{d}) = \nabla\mathcal{L}(\boldsymbol{\theta})^T\mathbf{d} + \frac{1}{2}\mathbf{d}^T(\mathbf{G}(\boldsymbol{\theta}) + \lambda\mathbf{I})\mathbf{d}$
  $\{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_N\} \leftarrow$ CG-MINIMIZE$(q_{\boldsymbol{\theta}}(\mathbf{d}), \mathbf{d}_0)$
  $\mathcal{L}_{\text{best}} \leftarrow \mathcal{L}(\boldsymbol{\theta} + \mathbf{d}_N)$
  for $i \leftarrow N-1, N-2, \ldots, 1$ **do** ▷ *backtracking*
    $\mathcal{L}_{\text{curr}} \leftarrow \mathcal{L}(\boldsymbol{\theta} + \mathbf{d}_i)$
    if $\mathcal{L}_{\text{prev}} \geq \mathcal{L}_{\text{best}} \wedge \mathcal{L}_{\text{curr}} \geq \mathcal{L}_{\text{best}}$ **then**
      $i \leftarrow i + 1$
      break
    $\mathcal{L}_{\text{best}} \leftarrow \mathcal{L}_{\text{curr}}$
  if $\mathcal{L}_{\text{prev}} < \mathcal{L}_{\text{best}}$ **then**
    $\lambda \leftarrow \frac{3}{2}\lambda$; $\mathbf{d}_0 \leftarrow \mathbf{0}$
    continue
  $\rho = (\mathcal{L}_{\text{prev}} - \mathcal{L}_{\text{best}})/q_{\boldsymbol{\theta}}(\mathbf{d}_N)$
  if $\rho < 0.25$ **then**
    $\lambda \leftarrow \frac{2}{3}\lambda$
  else if $\rho > 0.75$ **then**
    $\lambda \leftarrow \frac{3}{2}\lambda$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{d}_i$; $\mathbf{d}_0 \leftarrow \beta\mathbf{d}_N$; $\mathcal{L}_{\text{prev}} \leftarrow \mathcal{L}_{\text{best}}$

To perform distributed computation, we use a master/worker architecture in which worker processes distributed over a compute cluster perform data-parallel computation of gradients and curvature matrix-vector products and the master implements the Hessian-free optimization and coordinates the activity of the workers. All communication between the master and workers is via MPI. The master/worker architecture used in this paper is a simple one-layer architecture, with one master and many workers.

V. Application Enablement and Performance Tuning

In this section we describe our efforts to enable and tune the Parallel DNN Training application on the Blue Gene/Q computer system.

A. Matrix Multiplication on Blue Gene/Q

In order to optimize the performance of the matrix multiplication routine on Blue Gene/Q as well as to make it performance portable, the architectural levels are viewed in something of a telescoping fashion: node, core, and thread. As is typical on multicore architectures, decisions made at each level affect the decisions made at other levels as a result of feature interaction, so the division is not clean and optimizing locally tends to lead to implementations that are not globally optimal.

1) The Blue Gene/Q Architecture: As described in Section III, a Blue Gene/Q node consists of 16 PowerPC A2 compute cores operating at 1.6 GHz and, each, in turn, made up of 4 true hardware threads, fully in-order and single issue. Since each core can execute 4 wide SIMD FMA instructions, the floating point peak of a core is $8 \times 1.6 = 12.8$ GFLOPS, thus the theoretical peak operating speed of a node is 204.8 GFLOPS. When considering matrix-multiplication, this theoretical peak is (almost) achievable in practice for two primary reasons. First, because the caches and the memory itself are capable of supporting the bandwidth requirements of matrix multiplication. Second, because not every instruction in matrix multiplication is an FMA, it is important that multiple instructions, namely a floating point load along with an FMA, be executable in any given cycle. On Blue Gene/Q this is possible; while any single thread can only execute one instruction per cycle, two threads on the same core can execute (two) instructions at the same time as long as only one is performing a floating point arithmetic operation and the other is executing a different type of operation, such as a load or store.

2) A View from the Thread Level: At the lowest level of software construction there is the inner kernel for matrix multiplication. This code is written in assembly and is not threaded (the scaffolding for threading is encoded at a higher level of computational granularity). There are two overriding goals when writing code for the inner kernel. First, cover the latency to the appropriate level(s) in the memory hierarchy which equates to separating operand load from operand use by the appropriate number of cycles. Second, reduce the bandwidth to a level to which it can be fed by the cache(s) in which the operands will reside. For Blue Gene/Q, the inner (register-block) kernel consists of an 8×8 C matrix updated by a sequence of outer products. The A and B matrices are reformatted in such a way so as to allow strictly stride-one access to both matrices, thus the L1P prefetch engine is used, reducing the latency that must be covered.

Given that the code is written in assembly language, the scheduling of the instructions as well as the register layout can be made precise and latency coverage (approximately 20 cycles from the L1P prefetch unit) virtually guaranteed. Using higher-level languages such as C or Fortran leads to code that is easier to maintain, but may be subject to lack of SIMD instruction generation, suboptimal scheduling, and less aggressive register allocation by the compiler. Thus, changes in the compiler can lead to substantial changes in the generated code and the witnessed performance of computationally-intensive routines. It should be noted that the use of intrinsics, which have a one-to-one mapping to assembly level instructions, has been successfully employed in this domain on Blue Gene/Q[26].

3) Combining Threads: A View from the Core Level: While multiple instruction issue requires the use of only two hardware threads per core, for matrix multiplication, we used 4 hardware threads in all instances. This allows for the system to make more effective use of dual-issue capabilities. It also allows for us to get great leverage from the small (4 KB per thread) L1D cache via a novel software feature that we refer to as implicitly synchronized threads[27], [28].

Given the size of the L1D cache, the conventional practice of storing a slice of the A or B matrix in the cache is not viable unless the matrices have a very small K dimension. Using blocking on that matrix dimension results in multiple reads and writes of the C matrix, erasing potential performance gains. However, with precise control of task layout, data sharing can be improved in theory. Implicit synchronization allows us to realize this benefit in practice by minimizing thread skew (i.e. ensuring that all four threads execute at the same level of instruction throughput).

To illustrate, if we view the assignment of blocks of the resultant C matrix to be done in 2×2 sets to each core, we can see that both the A and B operand are shared across two threads. While this somewhat mitigates the cache splitting effect, the real benefit is somewhat different. With the threads cooperating to prefetch operand streams, the cache can be used as a staging area for the operands, as well as serving as the device through which the threads are coordinated. To see this, assume that the code is written (timed) so as to rely upon the property that every cache line needed is in the L1D cache before it is loaded in a register so as to be used in a computation. Further consider that for each thread, every other cache line of each operand is to be prefetched by a different (partner) thread. Thus, threads that advance past other threads are slowed down by the fact that half of their cache lines are "missing" and trailing threads can catch up. While it would seem that this scheme could break down should any thread get so far ahead of its compatriots that the operands that it prefetched are no longer in the L1D cache, higher levels of cache can be utilized in the same manner, greatly decreasing the odds of this occurring and extremely infrequent explicit synchronizations (via OpenMP and the special synchronization features of Blue Gene/Q) can be issued to eliminate this possibility as well as to orchestrate not just threads, but cores.

This shared prefetching, and the mutual dependency that it implies, enforces the implicit synchronization of threads primarily through the use of the L1D cache in this instance, but the method can be used at any level of the memory hierarchy and is not restricted to Blue Gene/Q systems. Furthermore, synchronizing and the cooperative use of shared operands, also lowers bandwidth requirements, as, with the correct task layout, a 2×2 grid, the 4 individual 8×8 outer product computations, each requiring 8 bytes/cycle (in double precision arithmetic) were they to run at 100% of peak, become one 16×16 outer product that requires only half that bandwidth, via a reduction in the surface to volume ratio.

4) Cooperation Between Cores: The Task and Node Levels: While the code for matrix multiplication is written so that it is correct regardless of the number of threads used per core or the number of cores used per node, it is tuned for the case where four threads per core are used and works best when, at the task level, the number of cores per rank is a perfect square. The square value allows for a (conceptually) square task layout, reducing the bandwidth from the (shared) L2 cache as much as is possible. This allows not only power savings, but the preloading of "to be used" operands with minimal impact on processing speed.

Two large levers that we have at the task level are the aforementioned explicit synchronization, made efficient through Blue Gene/Q's highly optimized OpenMP library, and a more hands-on control of allocated memory. Both of these techniques are used to gain close to peak performance at the expense of considerable care in coding and maintenance and are sometimes not done in more widely used libraries for those, and other, reasons. Explicit synchronization requires no real explanation; this is simply syncing all of the threads on a rank so that the cooperative properties mentioned above are established as a precondition. We tend to do this as new cache blocks are loaded, but the granularity of these synchronizations involves a trade-off pitting synchronization overhead against property guarantees. We manage memory by essentially keeping track of what we have allocated so that we can reallocate out of that memory instead of repeatedly freeing and allocating when new memory is required. This tends to enhance performance only slightly for large matrix operations, but it greatly reduces timing jitter which is invaluable when tuning a library or an application. While we constructed a rather involved method for controlling this allocated memory (the complexity stems from what has to be done when another application requests memory and cannot get it because our library is holding it), we did not use that method for the experiments seen here.

5) Bringing it Together: There are a several advantages to considering the system holistically, rather than constructing the code bottom-up (thread to node) or top-down. We have already mentioned the shared prefetching and the ability to lower the bandwidth requirements at both the node and core level by laying out the tasks to "slice up" the matrices by square "cookie cutters." Both of these properties require us to slice between levels of the system hierarchy. Knowledge of task layout also allows the implementer to have the threads cooperate for long- and short-term data prefetching. Long-term prefetching is used to pull in the next cache or register block of a matrix operand into the cache and is typically amortized over a large number of operations. As the C operand is not re-used in Level 3 BLAS operations like DGEMM, the intricate long-term prefetching of C matrix blocks proved important for the last 5% of performance gained. Short-term prefetching is the technique used in the inner kernel to hide latency. While latency hiding is greatly improved via the use of multiple threads on a core, cooperative prefetching has all of the benefits mentioned above and improves latency tolerance by allowing us to utilize more registers, per thread, for prefetching. This improves performance when new streams are started (the latency is higher when the L1P unit is not engaged) and when pulling new operand blocks from the L2 cache or main memory.

The tuning done specifically for this application included optimizations to maintain performance as the number of ranks per node is changed, handling small matrices and matrices with dimensions that do not lend themselves to full SIMDization without losing significant efficiency, and changes to the inner kernel to greatly improve the performance of calculations done using single-precision arithmetic.

B. Communication

In order to scale up the application, we abandoned the socket communication, the method used on a 64-node Intel/Linux cluster, and rewrote the application to utilize MPI communication. Socket communication requires the programmer take care of all details of communication which includes channel (communication parties) set up, data packaging and routing. MPI communication provides much more functionality when managing many nodes (of the order of thousands) in a large scale application and is portable, based on the MPI standard.

The Blue Gene/Q MPI communication library is heavily optimized, as MPI has wide support in high performance computing. The MPI-2.2 standard is supported on BG/Q with

the exception of a few features that are incompatible with the compute node kernel (e.g., those that require process forking). The optimized MPI on BG/Q is implemented on top of PAMI, a high-level active-message interface used to support many programming models.

The application previously used files to communicate among processes in order to synchronize the weights that are used in the DNN. This weight-synchronization step was converted to rely upon MPI, performance was improved by using the broadcast (MPI_Bcast) mechanism to take the advantage of the optimized MPI collectives available on the system.

C. Load Balance

One key factor affecting scalability is load balancing. At the processor/core level, efficiently overlapping computation and communication helps to improve the performance. At the application level, distributing data evenly cross compute nodes helps the program proceed in a synchronized pace to achieve better performance.

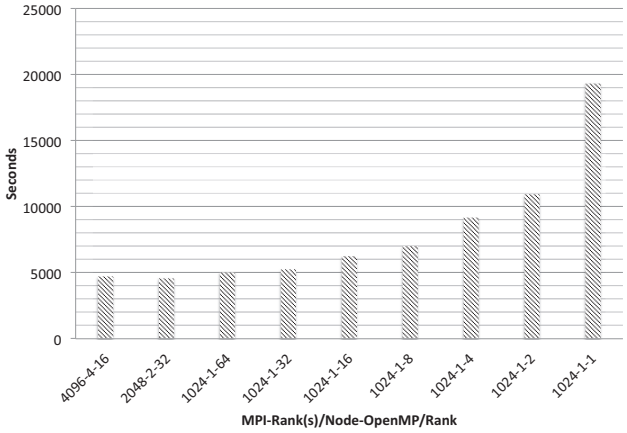
We distributed the data so as to minimize the run-time variation between workers. In speech recognition tasks, the training data comprises of several spoken utterances from thousands of speakers. These utterances in the training set are not all of the same length, so we preprocessed the data by sorting and computed the number of utterances per worker such that they all receive equal amount of data. This improves the load balancing and the effect is more apparent when the training data is scaled to larger sizes (millions of samples).

VI. Evaluation

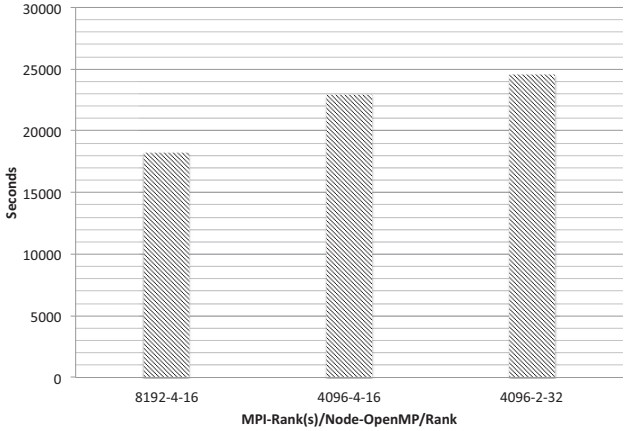
In this section, we first use up to 2 racks (2,048 nodes) of Blue Gene/Q and study the effect of varying MPI/OpenMP configuration on application performance. We also analyze the application to understand the manner in which the computation and (MPI) communication cycles are spent.

We start with a small sized (50-hour) input training data using 1 rack of Blue Gene/Q. 50 hrs of audio data amounts to roughly 18 million training samples. A plot of the time taken for each configuration is shown in Figure 1(a). As one might imagine, scaling up by increasing the number of OpenMP threads to fully utilize the cores improves the performance. As there are 16 cores on a node, one must use at least 16 threads to utilize all cores. This can be done by using 1 rank per node and 16 threads (1024-1-16) per rank or in any other combination that leads to the product of ranks per node and threads per rank being equal to 16. In our study, we target 64 threads per node. The potential configurations are as expected. As for the Rank(s)/Node and OpenMP ratio, the performance of 2048-2-32 is slightly better than 4096-4-16 which is better than 1024-1-64. Using more threads per core helps to hide the time gaps (e.g., stall cycles) for the hardware execution components.

Figure 1(b) illustrates how these runs scale with a larger input training data set comprising of 400-hours. An additional 22% speedup is obtained when the configuration is scaled to 8192-4-16 (two Blue Gene racks). A DNN on 400 hours can be trained using this configuration in 6.3 hours.



(a) 50-hour training data



(b) 400-hour training data

Fig. 1. Execution time for different configurations

Figures 2 and 3 plot the cycles spent in each of the key function calls of the algorithm described in Section IV.

We further analyze the cycles consumed by the master and the worker processes. Figure 2 and Figure 3 show the breakdown in cycles. The IU_Empty are the cycles where Instruction Unit is empty (e.g., Icache and Ierat Misses). AXU/FUX_Dep_Stalls are stall cycles due to dependency in Auxiliary/Fixed point units. The Committed Instructions are the cycles that the core is being "productive" doing the work. Figure 4 and Figure 5 are the cycles spent in collective and point-to-point MPI communication.

As the number of MPI ranks increases, shown in Figure 2, the master process needs to spend more time distributing the data (load_data) using point-to-point MPI calls and synchronizing the weights (sync_weights_master) using collective MPI calls (Figure 4(c). One can observe that for almost all function calls, as the MPI ranks increase, the computation time decreases (such as gradient_loss), while for other functions such as worker_curvature_product, the computation time can vary. This can be attributed to the fact that the algorithm randomly selects a small percentage of the data for this part of the computation which could contribute to the variance seen

in Figure 3.

TABLE I. SCALING UP PERFORMANCE

Training data	Intel Xeon 96 processes (hrs)	BG/Q 4096 MPI (hrs)	Speed Up	Frequency Adjustment
50-hour Cross-Entropy	9	1.3	6.9x	12.6x
50-hour Sequence	18.7	4.19	4.5x	8.2x

VII. DISCUSSION

To enable the application on a large scale, we investigate and improve the parallelism in different levels. At the core level, the multi-threading helps to overlap the empty cycles in the hardware components. The tuned matrix library helps to utilize the SIMD unit to have more floating point instructions executed in a single cycle. At the node level, load balancing cross MPI ranks synchronize the processes to improve the parallelization efficiency.

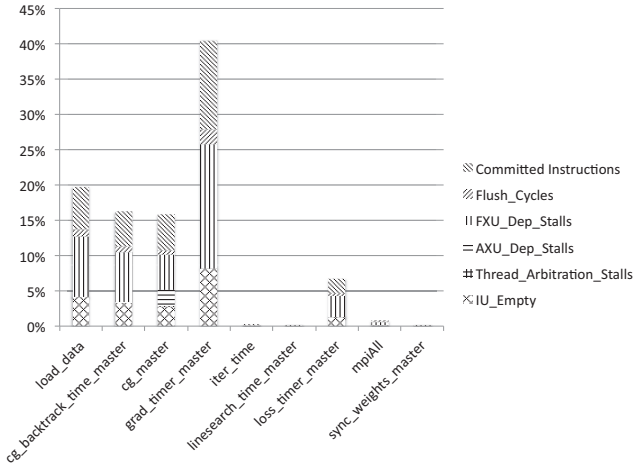
Analysis on our initial implementation indicated that the distribution of load (data) across the workers is critical in this application. Speech utterances are not of the same length, and therefore we needed to ensure that all workers processed the right set of utterances to adhere to the randomness needed by the algorithm as well as to ensure no load imbalances. This also made sure that the master did not wait on just one or two workers that had more data to work with.

It is key to point out that a Linux cluster that can be built with the same number of cores as used in Blue Gene will suffer from several communication bottlenecks (collisions), this is one of the main advantages of Blue Gene. A specialized network is needed to scale to that order of magnitude in order to take advantage of thousands of cores for this application.

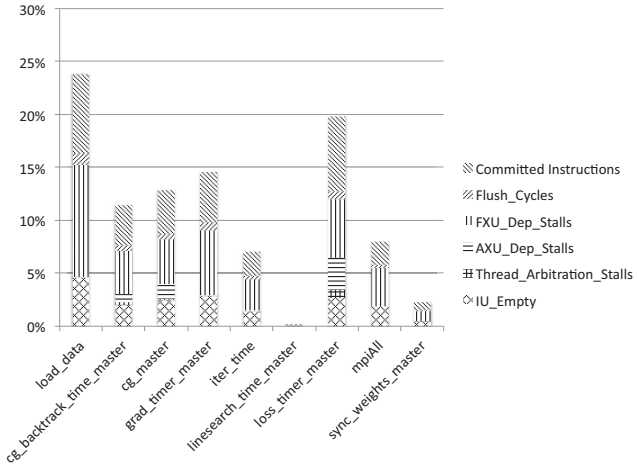
Table I summarizes the overall speed-ups obtained with the Blue Gene architecture. Allowing for the differences in core processor speed between an Intel-CPU based infrastructure and the Blue Gene computer system, we find that speed-ups of up to a factor of twelve can be obtained for two different algorithms, namely one that uses cross-entropy as the objective function and another that uses a discriminative criterion as the objective function to train the DNN. Both of these algorithms are extensively applied in speech applications, and this speed-up is significant, especially when a typical training set operates on several thousands of hours of data. Each of these networks converge to their optimal weights, after 20 to 40 iterations through the entire data set. As such, in order to turn around several experiments, performance is paramount. The last column in the figure adjusts for the clock frequency differences between the Intel (2.9 GHz) and Blue Gene processors (1.6GHz).

VIII. CONCLUSION

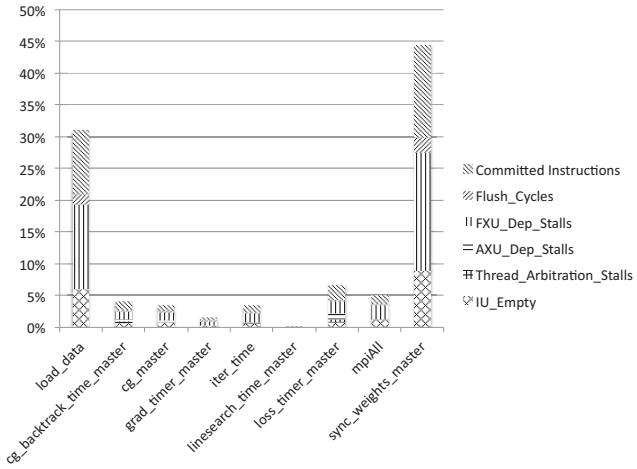
In this paper, we describe our efforts to enable Parallel Deep Neural Network Training on the IBM Blue Gene/Q (BG/Q) computer system. Specifically, we explore DNN training using the Hessian-free 2nd order optimization algorithm. We observe that we can achieve speed-ups that scale linearly



(a) 1024 MPI ranks, 1 rank/node, 64 OpenMP/rank

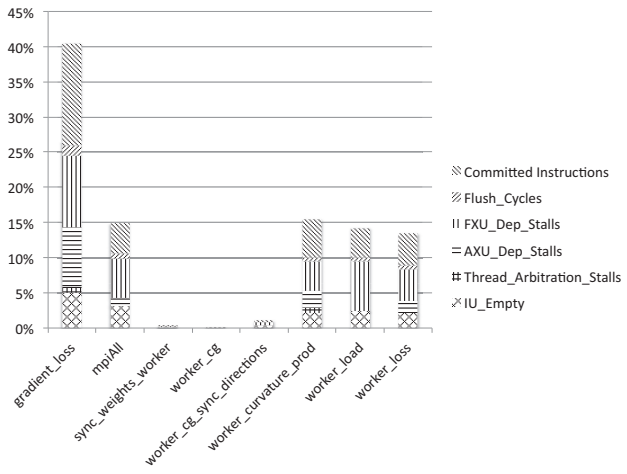


(b) 2048 MPI ranks, 2 ranks/node, 32 OpenMP/rank

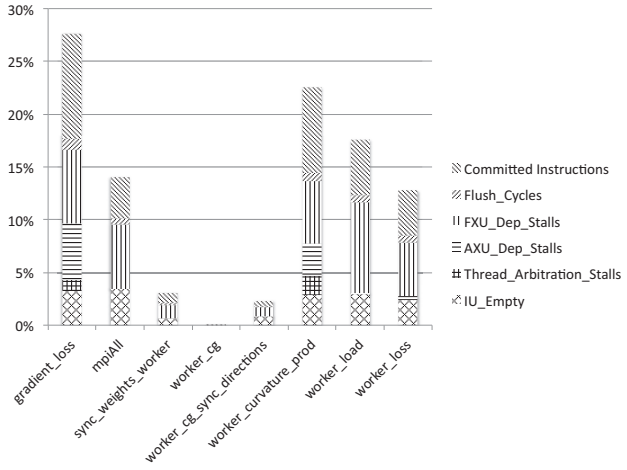


(c) 4096 MPI ranks, 4 ranks/node, 16 OpenMP/rank

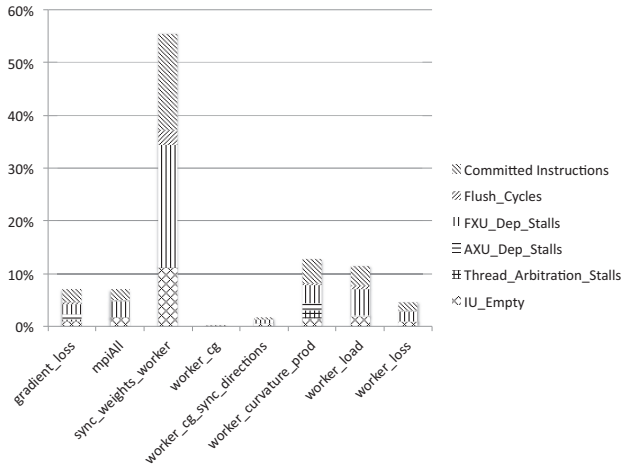
Fig. 2. Master process cycles break down



(a) 1024 MPI ranks, 1 rank/node, 64 OpenMP/rank

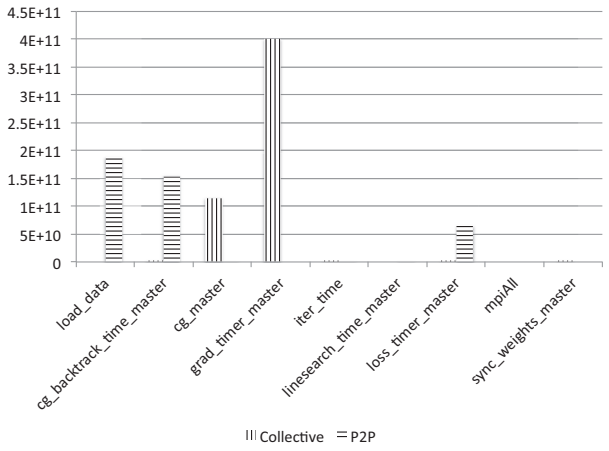


(b) 2048 MPI ranks, 2 ranks/node, 32 OpenMP/rank

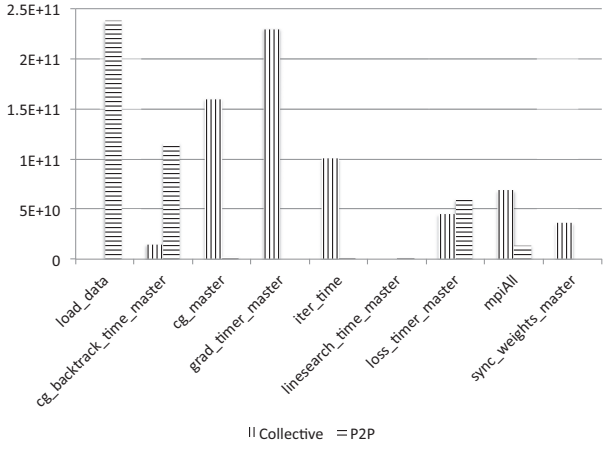


(c) 4096 MPI ranks, 4 ranks/node, 16 OpenMP/rank

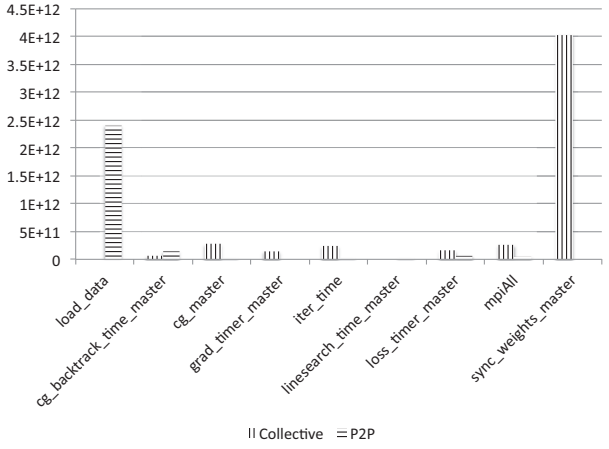
Fig. 3. Worker process cycles break down



(a) 1024 MPI ranks, 1 rank/node, 64 OpenMP/rank

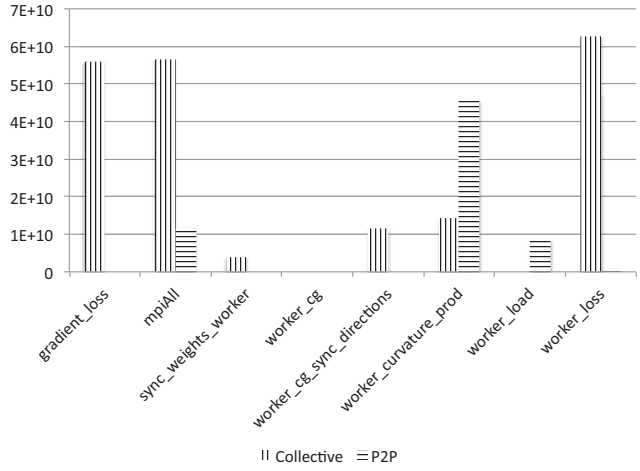


(b) 2048 MPI ranks, 2 ranks/node, 32 OpenMP/rank

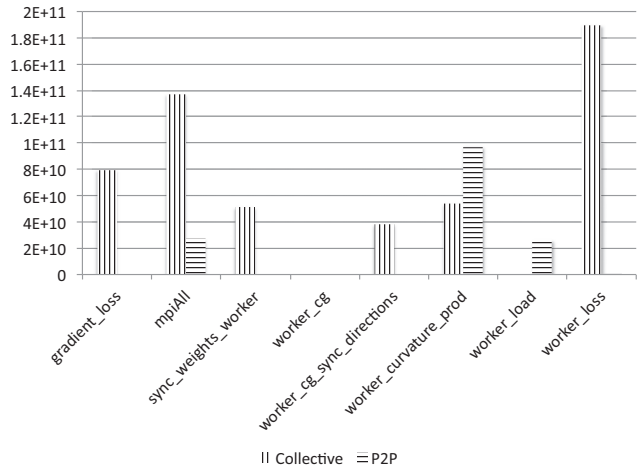


(c) 4096 MPI ranks, 4 ranks/node, 16 OpenMP/rank

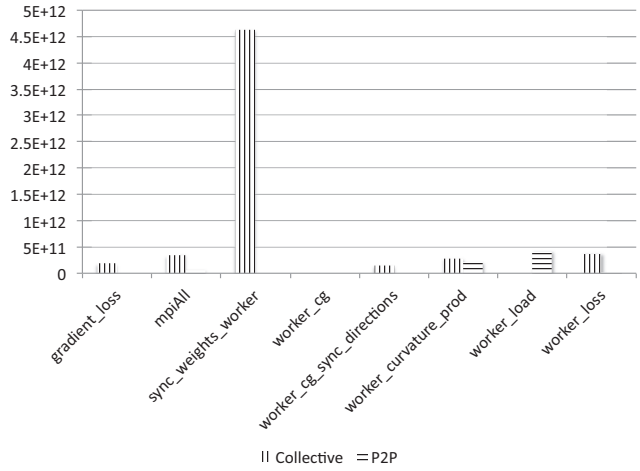
Fig. 4. Master MPI communication time



(a) 1024 MPI ranks, 1 rank/node, 64 OpenMP/rank



(b) 2048 MPI ranks, 2 ranks/node, 32 OpenMP/rank



(c) 4096 MPI ranks, 4 ranks/node, 16 OpenMP/rank

Fig. 5. Worker MPI communication time

up to 4096 processes. Beyond that, although we see a significant speed up, the speed improvements are sub-linear. We have demonstrated that with two racks of Blue Gene, we can train a deep network with over 100M parameters in 6 hours. This would take approximately a month to train on the Linux cluster of Intel-CPUs. Scaling to large data sets increases the communication costs (data exchanges) between the workers such that they dominate the overall training time. This requirement renders training with Intel-CPUs inefficient when dealing with large data sets. We have also observed on the larger training set (400 hours) that the BG/Q was able to achieve a good speedup, of the same order seen with the 50 hour dataset. We have demonstrated these gains in speed over two types of input, and by scaling the data to billions of training samples. In summary, these speed-ups obtained with BG/Q, make the training of large networks more feasible, and open up new avenues for research in compute-intensive applications. To date, the Hessian-free, second order optimization method with BG/Q provides the fastest training method and best WER for both, cross-entropy and sequence training objective functions on deep networks. Scalability of Blue Gene/Q stems from the lack of interference (e.g., noise, such as, OS jitter) from running compute processes. The system is essentially free of interference, which has been verified directly through measurements [29], [30]. From a financial perspective, Blue Gene/Q is also a leader in energy efficiency compared to the 30 different systems studied [31]. These factors further render use of Blue Gene/Q more apt for large-scale, big data applications.

The Blue Gene/Q has many more hardware features such as hardware transactional memory, thread level speculation and list prefetch engine. In the future, we plan to investigate the possibility of utilizing those hardware features to further improve the performance such that the application can handle an even larger input training data set in a reasonable amount of time.

ACKNOWLEDGMENT

We would like to express our appreciation for generous support and guidance from Fred Mintzer, John Magerlein, James Sexton and Michael Rosenfield.

REFERENCES

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] G. E. Hinton, S. Osindero, and Y. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. AISTATS*, 2010, pp. 249–256.

[4] N. Jaitly, P. Nguyen, A. W. Senior, and V. Vanhoucke, "Application Of Pretrained Deep Neural Networks To Large Vocabulary Speech Recognition," in *Proc. Interspeech*, 2012.

[5] B. Kingsbury, T. N. Sainath, and H. Soltau, "Scalable Minimum Bayes Risk Training of Deep Neural Network Acoustic Models Using Distributed Hessian-free Optimization," in *Proc. Interspeech*, 2012.

[6] F. Seide, G. Li, and D. Yu, "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks," in *Proc. Interspeech*, 2011.

[7] D. Yu, L. Deng, and G. E. Dahl, "Roles of Pre-Training and Fine-Tuning in Context-Dependent DBN-HMMs for Real-World Speech Recognition," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2010.

[8] T. N. Sainath, B. Kingsbury, B. Ramabhadran, P. Fousek, P. Novak, and A. Mohamed, "Making Deep Belief Networks Effective for Large Vocabulary Continuous Speech Recognition," in *Proc. ASRU*, 2011.

[9] Q. Le, J. Ngiam, A. Coates, A. Lahiri, B. Pronchnow, and A. Ng, "On Optimization Methods for Deep Learning," in *Proc. ICML*, 2011.

[10] J. Martens, "Deep learning via Hessian-free optimization," in *Proc. Intl. Conf. on Machine Learning (ICML)*, 2010.

[11] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized Stochastic Gradient Descent," in *Proc. NIPS*, 2010.

[12] G. E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines," Machine Learning Group, University of Toronto, Tech. Rep. 2010-003, 2010.

[13] T. N. Sainath and B. Kingsbury and B. Ramabhadran, "Improving Training Time of Deep Belief Networks Through Hybrid Pre-Training And Larger Batch Sizes," in *to appear in Proc. NIPS Workshop on Log-linear Models*, 2012.

[14] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *NIPS*, 2012.

[15] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-reduce for Machine Learning on Multicore," in *Proc. NIPS*, 2007.

[16] F. G. V. Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, R. A. van de Geijn, and F. Working Note #69, "Implementing level-3 BLAS with BLIS: Early experience," The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-13-03, 2013.

[17] A. A. Mirin, D. F. Richards, J. N. Glosli, E. W. Draeger, B. Chan, J.-L. Fattebert, W. D. Krauss, T. Oppelstrup, J. J. Rice, J. A. Gunnels, V. Gurev, C. Kim, J. Magerlein, M. Reumann, and H.-F. Wen, "Toward real-time modeling of human heart ventricles at cellular resolution: simulation of drug-induced arrhythmias," in *SC*, 2012, p. 2.

[18] B. Carnes, B. Chan, E. W. Draeger, J.-L. Fattebert, L. Fried, J. N. Glosli, W. D. Krauss, S. H. Langer, R. McCallen, A. A. Mirin, F. Najjar, A. L. Nichols, T. Oppelstrup, J. A. Rathkopf, D. F. Richards, F. H. Streitz, P. Vranas, J. J. Rice, J. A. Gunnels, V. Gurev, C. Kim, J. Magerlein, M. Reumann, and H.-F. Wen, "Science at llnl with ibm blue gene/q," *IBM Journal of Research and Development*, vol. 57, no. 1/2, p. 11, 2013.

[19] M. Gschwind, "Blue gene/q: design for sustained multi-petaflop computing," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 245–246. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304609

[20] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L.-T. Chiu, P. A. Boyle, N. H. Chist, and C. Kim, "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, 2012.

[21] R. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal, "On the use of stochastic Hessian information in unconstrained optimization," *SIAM Journal on Optimization*, vol. 21, no. 3, pp. 977–995, 2011.

[22] O. Vinyals and D. Povey, "Krylov subspace descent for deep learning," in *Proc. NIPS Workshop on Optimization and Hierarchical Learning*, 2011.

[23] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994.

[24] N. N. Schraudolph, "Fast curvature matrix-vector products for second-order gradient descent," *Neural Computation*, vol. 14, pp. 1723–1738, 2004.

[25] B. Kingsbury, "Lattice-based optimization of sequence classification criteria for neural-network acoustic modeling," in *Proc. ICASSP*, 2009, pp. 3761–3764.

[26] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Opportunities for parallelism in matrix multiplication. FLAME Working Note #71," Department of Computer Science, The University of Texas at Austin, Tech. Rep. TR-13-20, 2013, submitted to IPDPS2014.

[27] A. E. Eichenberger and J. A. Gunnels, "Shared prefetching to reduce execution skew in multi-threaded systems," Jul 2013. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1087835

[28] J. Gunnels, "Making good enough...better: Addressing the multiple objectives of high-performance parallel software with a mixed global-local worldview," 2012, slides of a talk given at ICERM, Synchronization-reducing and Communication-reducing Algorithms and Programming Models for Large-scale Simulations. [Online]. Available: http://icerm.brown.edu/materials/Slides/tw-12-1/Making_Good_Enough...Better-Addressing_the_Multiple_Objectives_of_High-Performance_Parallel_Software_with_a_Mixed_Global-Local_Worldview_%5D_John_A_Gunnels,_IBM.pdf

[29] K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini, "A performance and scalability analysis of the BlueGene/L architecture," in *Proc. IEEE/ACM SC04*, 2004.

[30] K. Singh, E. Ipek, S. McKee, B. deSupinski, M. Schulz, and R. Caruana, "Predicting parallel application performance via machine learning approaches," in *Concurrency and Computation: Practice and Experience*, 2006.

[31] http://green500.org/.